WPI-CS-TR-00-25 December 2000, updated March 2002

Incremental Maintenance of Schema-Restructuring Views in *SchemaSQL*

by

Andreas Koeller
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Incremental Maintenance of Schema-Restructuring Views in *SchemaSQL*

Andreas Koeller and Elke A. Rundensteiner
Computer Science Department, Worcester Polytechnic Institute
{koeller, rundenst}@cs.wpi.edu

March 6, 2002

Abstract

The integration of data, especially from heterogeneous data sources (ISs), is a hard and widely studied problem. One particularly challenging issue is the integration of sources that are semantically equivalent (i.e., whose states can be mapped onto each other by an isomorphism) but schematically heterogeneous. While two such data sources may represent the same information, one may store the information inside tuples (*data*) while the other may store it in attribute or relation names (*schema*). The *SchemaSQL* query language is a recent solution to this problem powerful enough to restructure such sources into each other without the loss of information. However, the issue of maintenance of *SchemaSQL* views, once materialized over semantically heterogeneous sources, has not yet been addressed. In this paper, we propose an incremental view maintenance strategy for such schema-restructuring views. Our strategy based on an algebraic representation of the view query correctly transforms a data update or a schema change to a source into data updates, schema changes, and even mixed sequences of schema and data update requests to be applied to the view. We give a proof of the correctness of the strategy. We have also developed a prototype implementation of both a *SchemaSQL* query processor and a *SchemaSQL* View Maintainer, and then used it to compare the performance of incremental view maintenance versus complete view recomputation for this new class of schema-restructuring views. We describe both the implementation and the experiments in the paper and conclude that in many cases incremental view maintenance in *SchemaSQL* is significantly faster than recomputation.

Keywords: Heterogeneous Databases, Materialized Views, SchemaSQL, Incremental View Maintenance, Schema Restructuring.

1 Introduction

Information sources, especially on the Web, are increasingly independent from each other, being designed, administered and maintained by a multitude of autonomous data providers. Issues in data integration include the heterogeneity of data and query models across different sources, called model heterogeneity [FRV95, GRVB98, HGMI$^+$95] and incompatibilities in schematic representations of different sources even when using the same data model, called schema heterogeneity [MIR93, LSS96]. Overcoming these problems is critical in achieving integration of a wide variety of information sources. Much work on these problems has dealt with the integration of schematically different sources under the assumption that all "data" is stored in tuples and all "schema" is stored in attributes. We will focus on another aspect of this issue in our paper, namely on the integration of heterogeneous sources under the assumption that schema elements may express data and vice versa.

One recent promising approach at overcoming such schematic heterogeneity is the language *SchemaSQL*, an SQL-extension devised by Lakshmanan et al. [LSS96, LSS99]. *SchemaSQL* allows to soften the distinction between schema and data in the relational data model by allowing to query schema (such as lists of attribute or relation names) in SQL-queries and also to use sets of values obtained from *data tuples* as *schema* in the output relation. This concept leads to a versatile query language which among other features allows to transform semantically equivalent but syntactically different schemas [LSS96, Mil98] into each other. Similar to SQL-views, *SchemaSQL*-views can be used to transform relational databases into whatever format is required by a (relational) data integration system. Therefore, *SchemaSQL* makes it possible to include a larger class of information sources into an information system.

However, the issue of view maintenance in such a system is still open. View maintenance in a *SchemaSQL* view is non-trivial, especially since such views can transform data into schema and vice versa as illustrated below. In this paper, we present the first incremental maintenance strategy for *SchemaSQL*. The strategy works correctly not only under data updates, but also under schema changes.

1.1 Motivating Example

Fig. 1 gives an example of a *SchemaSQL* query to demonstrate the capabilities of this language. Note that the two relational schemas in Fig. 1 are able to hold the same information and can be mapped into each other using *SchemaSQL* queries. The view query restructures the input relations on the left side representing

BA

Destination	Business	Economy
Paris	1200	600
London	1100	475

LH

Destination	Business	Economy
Paris	1220	700
London	1180	500

⇒

```
create view CITY(Type, AIRLINE) AS
 select PRICETYPE, FLIGHT.PRICETYPE
 from
   -> AIRLINE,
   AIRLINE FLIGHT,
   AIRLINE-> PRICETYPE,
   FLIGHT.Destination CITY
 where PRICETYPE <> 'Destination'
 and FLIGHT.PRICETYPE <= 1100;
```

⇒

LONDON

Type	BA	LH
Business	1100	*null*
Economy	475	500

PARIS

Type	BA	LH
Economy	600	700

Figure 1: A *SchemaSQL* Query and its Effects.

airlines into attributes of the output relations on the right side representing destinations. The *arrow*-operator (`->`) attached to an element in the FROM-clause of a *SchemaSQL*-query allows to query schema elements, giving *SchemaSQL* its meta-data restructuring power. Standing by itself, it refers to "all relation names in that database", attached to a relation name it means "all attribute names in that relation".

SchemaSQL is also able to transform data into schema. For example, *data* from the attribute Destination in the input schema is transformed into *relation names* in the output schema, and vice versa *attribute names* in the input (Business and Economy) are restructured into *data*.

Now consider an update to one of the base relations in our example. Let a tuple $t(\mathsf{Destination} \Rightarrow \mathsf{Berlin}, \mathsf{Business} \Rightarrow 1400, \mathsf{Economy} \Rightarrow 610)$ be added to the base table LH (a **data update**). The change to the output would be the addition of a new relation Berlin (a **schema change**) with the same schema as the other two relations. This new relation would contain one tuple $t(\mathsf{Type} \Rightarrow \mathsf{Economy}, \mathsf{BA} \Rightarrow null, \mathsf{LH} \Rightarrow 610)$. In this example, a data update is transformed into a schema change, but all other combinations are also possible. The effect of the propagation of an update in such a query depends on numerous factors, such as the input schema, the view definition, the set of unique values in the attribute Destination across *all* input relations (city names), and the set of input relations (airline codes). For example, if a value Berlin already existed in one of the input tables, the propagation would also depend on whether other airlines offer a flight to Berlin in the Economy-class.

In summary, a schema-restructuring view must be able to propagate arbitrary data updates and schema changes that occur in its input. In the process of propagation, any such type of change could be transformed into any other in the view. This situation is significantly more complicated than update propagation in the common SQL views [BLT86, QW91, GMS93, GL95, MKK97, et al.] which deals only with propagation of data updates that are never transformed into schema changes. This problem, to our knowledge, has not been studied before.

1.2 Contributions

In the context of schema-restructuring views, there are several new issues that we must address. First, it is not sufficient to consider data updates (DUs) for *SchemaSQL*, but also schema changes (SCs). Also, *SchemaSQL* views can transform schema into data and vice-versa, thus requiring a framework that can propagate $\{DU|SC\}^* \rightarrow \{DU|SC\}^*$. As shown in this paper, using the standard approach of generating query expressions that compute some kind of "delta" relation Δ between the old and the new view after an update is not possible for *SchemaSQL*, since the schema of Δ would not be defined.

The contributions of this paper are as follows: (1) we give an algebra-based solution to the problem of incremental view maintenance of schema-restructuring views defined in *SchemaSQL*, (2) we prove this approach correct by a method similar to the equational reasoning given in [GL95], (3) we present a prototype implementation of a query processor for a subset of *SchemaSQL* and an incremental view maintenance system in Java over JDBC-capable databases, and (4) we describe experiments we have conducted on our implementation to gain insights into the performance of our algorithm.

1.3 Outline of Paper

Section 2 reviews some background on *SchemaSQL*, in particular the additional algebra operators used in *SchemaSQL* evaluation, Section 3 explains our view maintenance strategy and Section 4 proves correctness of our approach. Section 5 gives a brief overview over our implementation. Section 6 shows the results of our performance experiments. Finally, Sections 7 and 8 give related work and conclusions, respectively.

2 Background

2.1 Notation

A *value* is an element of data that is stored in a relation. Examples include strings, numbers, and dates. A *domain* D is a set of *values*.[1] D_N is the special domain of "attribute- and relation names". We implicitly assume that there is a bijective mapping from some domains to D_N. This means that the values of some, but not necessarily all, domains can be converted to names and vice versa.

A *relation* is a 3-tuple $R = (n, S, E)$ with $n \in D_N$ (the relation name), $S = (a_1, a_2, \ldots, a_n) \in (D_N)^n$ (the schema—a tuple of n attribute names) and $E \subseteq \{D_1 \times D_2 \times \ldots \times D_n\}$ (the relation extent, which is a subset of the cross-product of the domains $D_1 \times D_2 \times \ldots \times D_n$). Note that this definition associates exactly one value in S with each domain from which E is constructed (the name of an "attribute").

A *relational tuple* $t \in E$ is an n-tuple and is an element of a relation's extent. An operator $t[a_{l_1}, a_{l_2}, \ldots, a_{l_k}]$ returns the projection of t on the attributes named $a_{l_1}, a_{l_2}, \ldots, a_{l_k}$. We also define $t[*\backslash\{a_1, \ldots, a_n\}]$ to be the projection of t onto all its attributes except the ones named $a_1, \ldots, a_n$.

An *attribute* $A_i \subseteq D_i$ is a multiset that is constructed as follows: $A_i = \{t[a_i] \mid t \in E\}$, or short $A_i = E[a_i]$. Then *attribute* A_i has *attribute name* a_i. Note that we denote *attributes* by capital letters (as they are sets) and *attribute names* by small letters. We extend this notation for $E[a_1, \ldots, a_k]$ to mean the *projection* of extent E on the attributes $A_1, \ldots, A_k$. For readability, if we refer to attribute A of $R = (n, S, E)$, we actually mean the pair $A = (a, E[a])$, with $a \in S$. The term *prime attribute* refers to an attribute that is a member of any key of R and the term *non-prime attribute* refers to an attribute that is not a member of any key of R (cf. [Ull89]). The *distinct*-operator $\langle a_i \rangle$ on an attribute A_i in extent E returns the set of distinct values in A_i by removing all duplicates from the multiset $E[a_i]$.

Functional dependencies in R are defined as usual (cf. [Ull89, Chapter 7]), with $X \rightarrow A$ defining the attribute A to be functionally dependent on the set of attributes X (i.e., for any $t \in E$, the value of $t[a]$ depends only on $t[x_1, \ldots, x_k]$) . Likewise, we assume the usual definitions of *natural join* $\bowtie$ and *cross product* $\times$.

2.2 *SchemaSQL*

In relational databases it is possible to store equivalent data in different schemas, as Miller et al. [MIR93] have shown. It is also possible under certain conditions to transform data in such schemas into each other without the loss of information [Mil98]. *SchemaSQL* is an SQL derivative designed by Lakshmanan et al. [LSS96] which can be used to achieve such schema transformations.

In [LSS99], Lakshmanan et al. describe an extended algebra and algebra execution strategies to implement a *SchemaSQL* query evaluation system. It extends the standard SQL algebra which uses operators such as $\sigma(R)$, $\pi(R)$, and $R \bowtie S$ by adding four operators named UNITE, FOLD, UNFOLD, and SPLIT originally introduced by Gyssens et al. [GLS96]. Lakshmanan et al. show that any *SchemaSQL* query can be translated into this extended algebra.

We now define the four operators used in SchemaSQL in a concise manner. Examples for the four operators defined in this section can be found in Fig. 2. We will refer to the input relation of each operator as R and to the output relation as Q.

[1]Throughout this paper, we will use capital letters R to denote *(multi)sets* and small letters a to denote elements of sets.

BA

Destination	Business	Economy
Paris	1200	600
London	1100	475

LH

Destination	Business	Economy
Paris	1220	700
London	1180	500

$\Downarrow$ UNITE $_{\text{Airline}}$

TMP_REL_0001

Airline	Destination	Business	Economy
BA	Paris	1200	600
BA	London	1100	475
LH	Paris	1220	700
LH	London	1180	500

$\Downarrow$ FOLD $_{\text{Type, Price, \{Business,Economy\}}}$

TMP_REL_0002

Airline	Type	Destination	Price
BA	Business	Paris	1200
BA	Business	London	1100
BA	Economy	Paris	600
BA	Economy	London	475
LH	Business	Paris	1220
...	...	...	...

$\Longrightarrow$
STANDARD-SQL

```
select *
from tmp_rel_0002
where price <=
1100;
```

TMP_REL_0003

Airline	Type	Destination	Price
BA	Business	London	1100
BA	Economy	Paris	600
BA	Economy	London	475
...	...	...	...

$\Uparrow$ UNFOLD $_{\text{Airline, Price}}$

TMP_REL_0004

Type	Destination	BA	LH
Business	London	1100	*null*
Economy	Paris	600	700
Economy	London	475	500

$\Uparrow$ SPLIT $_{\text{Destination}}$

LONDON

Type	BA	LH
Business	1100	*null*
Economy	475	500

PARIS

Type	BA	LH
Economy	600	700

Figure 2: The Four *SchemaSQL* Operators UNITE, FOLD, UNFOLD, SPLIT.

2.2.1 UNITE

This operator is defined on a set of k relations $R^* = \{R_1, \ldots, R_k\}$ with a_p as an argument. We define a set $N^* = \{n_{R_1}, \ldots, n_{R_k}\}$, which is the set of all relation names in R^*, in the new domain D_p. We further denote by N_k the relation $R(n, E, S)$ with $N = n_k$. Then, for each R_i, we assume $S_{R_i} = (a_1, \ldots, a_n)$ and $E_{R_i} \subseteq \{D_1 \times \ldots \times D_n\}$. Note that this implies that all R_i have the same schema. The output of the UNITE operator is then one relation $Q = \text{UNITE}_{a_p}(R^*)$ with:

$E_Q \subseteq \{D_1 \times \ldots \times D_n \times D_p\}$ and $S_Q = (a_1, \ldots, a_n, a_p)$ with $E_Q = \bigcup_{n_k \in N^*} (N_k \times \{n_k\})$. Note that N_k appears both as a *relation* (N_k) and a *relation name* (n_k). In words, a new relation is constructed by taking the union of all input relations and adding a new attribute A_p whose values are the *relation names* of the input relations.

In Fig. 2, the UNITE-operator is defined over the set of relations BA, LH and has the attribute name Airline as its argument.

2.2.2 FOLD

The FOLD-operator works on a relation $R = (n_R, E_R, S_R)$ with

$$E_R \subseteq \{D_1 \times \ldots \times D_n \times \underbrace{D_d \times \ldots \times D_d}_{k \text{ times}}\}$$

and $S_R = (a_1, \ldots, a_n, a_{n+1}, \ldots a_{n+k})$ and takes as arguments the *names* of the pivot and data attributes a_p and a_d in its output relation. Note that this definition requires that k attributes of R have to be of the same domain. With R having $n + k$ attributes, we then define:

$Q = \text{FOLD}_{a_p, a_d}(R) = (n_Q, E_Q, S_Q)$ with $n_Q = n_R$, $E_Q \subseteq \{D_1 \times \ldots \times D_n \times D_d \times D_p\}$ and $S_Q =$

$(a_1, \ldots, a_n, a_d, a_p)$. We define $A^* = \{a_{n+1}, \ldots a_{n+k}\}$ as a set of values in a new domain D_p where the values are obtained by the above-mentioned conversion of attribute names into data values. Finally, $E = \bigcup_{a_k \in A^*} (R[a_1, \ldots, a_n, a_k] \times \{a_k\})$. In words, the operator takes all data values from the set of related attributes, and sorts them into *one* new attribute a_d, introducing another new attribute a_p that holds the former attribute names. Note that, since a_p becomes part of a key for a_d, it has to be included in the set X for any functional dependency $X \rightarrow A_d$.

In Fig. 2, the FOLD-operator is defined on relation TMP_REL_0001 and has the arguments $a_p =$ Type, $a_d =$ Price, $A^* = \{$Business,Economy$\}$.

2.2.3 UNFOLD

The UNFOLD-operator is the inverse of FOLD.

The UNFOLD-operator on a relation $R = (n_R, E_R, S_R)$ with $E_R \subseteq \{D_1 \times \ldots \times D_n \times D_p \times D_d\}$ and $S_R = (a_1, \ldots, a_n, a_p, a_d)$ takes two arguments a_p, a_d which are attribute names from S_R. To simplify the notation and without loss of generality, we reorder the attributes in R (by exchanging the indices on both S_R and E_R accordingly), such that A_p and A_d become the last two attributes in R. We call A_p the *pivot attribute* and A_d the *data attribute*. Let R have $n+2$ attributes. Let us further impose two conditions on functional dependencies in R: $(X \rightarrow Y) \Rightarrow A_d \notin X$ and $\exists(X \rightarrow A_d)$ with $A_p \in X$. That is, A_d must be non-prime, A_p must be prime and A_d must depend on A_p. We also set $A^* = R\langle A_p \rangle$ (the set of distinct values in A_p), $k = |A^*|$ and impose a total order on A^* to assign an index $1 \leq i \leq k$ to each of its elements $(A^* = \{a_1^*, \ldots, a_k^*\})$.

Then, $Q = \text{UNFOLD}_{a_p,a_d}(R) = (n_Q, S_Q, E_Q)$ with $N_Q = n_R$,

$$E_Q \subseteq \{D_1 \times \ldots \times D_n \times \underbrace{D_d \times \ldots \times D_d}_{k \text{ times}}\}$$

and $S_Q = (a_1, \ldots, a_n, a_1^*, \ldots, a_k^*)$. The extent is constructed by $E_Q = E_R[a_1, \ldots, a_n] \bowtie E_1 \bowtie \ldots \bowtie E_k$ with $E_i = \{t[a_1, \ldots, a_n, a_d] \mid t \in E_R \ \wedge \ t[a_p] = a_i^*\}$.

In words, the schema of Q consists of all attributes in R except the data and pivot attribute, plus one attribute for each distinct data value in the pivot attribute. Then, each tuple t' in Q is constructed by taking a tuple t in R and filling each new attribute A_i with the value from attribute A_d in a tuple from R that has the *name* a_i as *value* in A_p (assuming an implicit conversion between names and values as required above). The new attributes all have the domain D_d of the old attribute A_d. Note that the requirement for a_p to be prime is not explicit in Lakshmanan's original operator. This is a design decision that helps to clarify the semantics of the UNFOLD/FOLD-operator pair, ensuring the output relation for UNFOLD to be in 1NF.

In Fig. 2, the UNFOLD-operator is defined over relation TMP_REL_0003 and takes as its arguments $a_p =$ Airline and $a_d =$ Price. The operator then produces output by taking tuples from TMP_REL_0003, and filling the attributes representing airlines with values from the data attribute Price in TMP_REL_0003, matching attribute names in the output relation with the values of the pivot attribute Airline in the input relation.

2.2.4 SPLIT

The SPLIT-operator is the inverse of the UNITE-operator. It transforms a single relation $R = (n_R, E_R, S_R)$ with $E_R \subseteq \{D_1 \times \ldots \times D_n \times D_p\}$ and $S_R = (a_1, \ldots, a_n, a_p)$ into a *set* of k relations with the same schema.

It takes as argument the name of the pivot attribute a_p which we assume to be the last in R. We require that A_p does not have NULL-values, i.e., $\forall x \in A_p : x \neq \perp$. The output of SPLIT is a set of relations $Q^* = \textsc{Split}_{a_p}(R) = \{Q_1, \ldots, Q_k\}$ with $A^* = R\langle A_p \rangle$ and $k = |A^*|$. We will refer to the ordered elements of A^* as in the UNFOLD-case, i.e., $A^* = \{a_1^*, \ldots, a_k^*\}$. For each output relation Q_i, we have:

$n_{Q_i} = a_i^*$, $E_{Q_i} \subseteq \{D_1 \times \ldots \times D_n\}$, $S_{Q_i} = (a_1, \ldots, a_n)$, and $E_{Q_i} = \{t[a_1, \ldots, a_n] \mid t \in R \wedge t[a_p] = a_i^*\}$. In words, we break down R into k relations of the same schema, with the new relation names the k distinct values from R's attribute A_p.

In Fig. 2, the SPLIT-operator is defined over relation TMP_REL_0004, takes as its only argument a_p = Destination, and produces 2 tables names LONDON and PARIS.

2.2.5 Flat Schemas

As suggested by the example in Fig. 1, existing relational schemas are often built with data-carrying attribute- and relation-labels, which is an important reason for schematic heterogeneity [LSS96]. In the relational model, relation and attribute labels are assumed to not contribute to the semantic content of the relation—a principle that is often misunderstood in database design. Lakshmanan et al. show that, while there are many representations of a real-world concept in the relational model, only some such schemas will not carry semantic information in their attribute and relation names. They give a definition for the notion of such *flat schemas* which we will only use in an informal way in this paper.

Definition 1 (Flat Schema) *Assume infinite pairwise disjoint sets of names $\mathcal{N}$ and values $\mathcal{V}$. Let* dom : $\mathcal{N} \to 2^{\mathcal{V}}$ *be a partial function such that for each $n \in \mathcal{N}$, whenever* dom(n) *is defined, it associates name n with a non-empty set of values* dom$(n) \subset \mathcal{V}$. *Then, a relation schema $R(A_1, \ldots, A_n)$ is said to be* flat *iff all the entries $R, A_1, \ldots, A_n$ are names. A database schema is flat if all relation schemas in it have this property.*

With the notion of schema equivalence presented in [MIR93], "flattening" a relation refers to the process of transforming a relation into its flat schema-equivalent relation, preserving the information capacity of the relation.

1. If the input schema consists of $n > 1$ tables of the same schema, apply a UNITE-operator with the name of the "pivot"-attribute as a parameter to obtain relation R' (i.e., $S \xrightarrow{\textsc{Unite}} R'$).

2. If R or R' contains a set of attributes $\{u_1, \ldots, u_n\}$ whose labels are not from $\mathcal{N}$, apply a fold operator on $\{u_1, \ldots, u_n\}$, with the names of the two resulting attributes B, C as parameters (i.e., $R' \xrightarrow{\textsc{Fold}} R''$).

2.2.6 *SchemaSQL* Query Evaluation

Similar to traditional SQL evaluation, [LSS99] proposes a strategy for *SchemaSQL* query evaluation that first constructs and then processes an algebra query tree. In that way, *SchemaSQL* can be efficiently implemented over an SQL database system, which Lakshmanan et al. have shown in [LSS99].

For simplicity of notation, we will treat a database consisting of n relations $R_1, \ldots, R_n$ of the same schema as a relation and denote it by R.

In order to evaluate a *SchemaSQL* query, an algebra expression using standard relational algebra plus the four operators introduced above is constructed. This expression is of the following form [LSS99]:

$$V = \text{SPLIT}_a(\text{UNFOLD}_{b,c}(\pi_{\bar{d}}(\sigma_{cond}(\text{FOLD}_{e_1,f_1,\bar{g_1}}(\text{UNITE}_h(R_1)) \times \ldots \times \text{FOLD}_{e_m,f_m,\bar{g_m}}(\text{UNITE}_h(R_m)))))) \quad (1)$$

with attribute names a, b, c, e_i, f_i, h_i, the sets of attribute names $\bar{d}$ and $\bar{g}_i$, and selection predicates *cond* determined by the query. Any of the four *SchemaSQL* operators may be missing from the expression (i.e., may not be needed for a particular query). $R_1 \ldots R_m$ are base relations, or, in the case that the expression contains a UNITE-operator, sets of relations with equal schema.

The algebraic expression for our running example (Fig. 1) is:

$$\begin{aligned} V \quad = \quad & \text{SPLIT}_{\text{Destination}}(\text{UNFOLD}_{\text{Airline,Price}}(\sigma_{\text{Price}<1100}(\\ & \text{FOLD}_{\text{Type, Price, }\{\text{Business,Economy}\}}(\text{UNITE}_{\text{Airline}}(\text{BA,LH}))))) \end{aligned} \quad (2)$$

This algebraic expression is then used to construct an algebra tree (Fig. 3) whose nodes are any of the four *SchemaSQL* operators or a "Standard-SQL"-operator (including the π, σ, and $\times$-operators of the algebra expression) with standard relations "traveling" along its edges. The query is then evaluated by traversing the algebra tree and executing a query processing strategy for each operator, analogous to traditional SQL query evaluation.

Note that the query tree could include $\times$-operators (which do not exist in our example), but that the order of UNITE, FOLD, UNFOLD, SPLIT (if they exist) is fixed by the template in Equation 1. The UNITE operator takes a number of relations of the same schema as an input, while the SPLIT-operator produces as output a set of relations of the same schema. Note that the algebra tree in Fig. 3 is very simple, in more complex queries, the tree could "fork" at the *Standard-SQL-node*, and several smaller "flattening" trees using UNITE- and FOLD-operators could occur. In that case, and also in the case of standard relational joins, the *Standard-SQL-node* would itself contain a more complex algebra tree containing simple SQL algebra nodes.

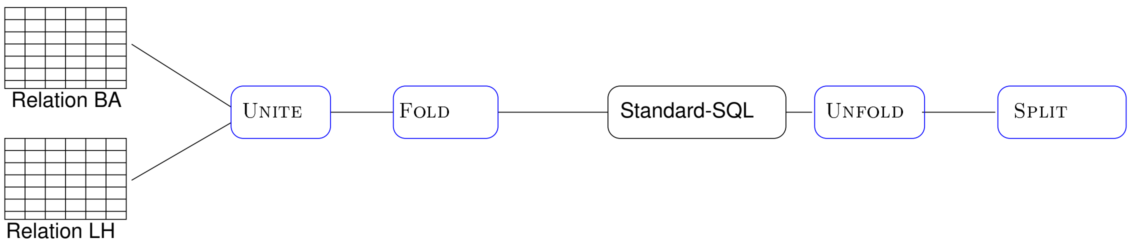

Figure 3: The Algebra Tree for the Example in Fig.1

3 The *SchemaSQL* Update Propagation Strategy

3.1 Classes of Updates and Transformations

The updates that can be propagated through *SchemaSQL* views can be grouped into two categories: *Schema Changes (SC)* and *Data Updates (DU)*. Schema changes that we consider are: *add-relation*(n, S), *delete-relation*(n), *rename-relation*(n, n') with relation names n, n' and schema S as introduced in Section 2.1 and *add-attribute*(r, a), *delete-attribute*(r, a), *rename-attribute*(r, a, a') with r the name of the relation R that the attribute named a belongs to, a' the new attribute name in the rename-case, and the notation otherwise as above. Data updates are any changes affecting a tuple (and not the schema of the relation), i.e., *add-tuple(r,t), delete-tuple(r,t), update-tuple(r,t,t'))*, with t and t' tuples in relation R with name r. Note that we consider *update-tuple* as a basic update type, instead of breaking it down into a *delete-tuple* and an *add-tuple*. An *update-tuple* update consists of two tuples, one representing an existing tuple in R and the other representing the values of that tuple after the update. This allows to keep relational integrity constraints valid that would otherwise be violated temporarily.

3.2 *SchemaSQL* Update Propagation vs. Relational View Maintenance

Update propagation in *SchemaSQL*-views, as in any other view environment, consists in recording updates that occur in the input data and translating them into updates to the view extent. In incremental view maintenance of SQL views [QW91, GL95, MKK97], many update propagation mechanisms have been proposed. Their common feature is that the new view extent is obtained by first computing *extent differences* between the old view V and the new view V' and then adding them to or subtracting them from the view, i.e., $V' = (V \backslash \nabla V) \cup \Delta V$, with ∇V denoting some set of tuples computed from the base relations that needs to be deleted from the view and ΔV some set that needs to be added to the view [QW91].

In *SchemaSQL*, this mechanism leads to difficulties. If *SchemaSQL* views must propagate both schema *and* data updates, the schema of ΔV or ∇V does not necessarily agree with the schema of the output relation V. But even when considering only data updates to the base relations, the new view V' may have a different schema than V. That means the concept of set difference between the tuples of V' and V is not even meaningful. Thus, we must find a way to incorporate the concept of schema changes. For this purpose, we now introduce a data structure ∂ which represents a sequence of n data updates DU and schema changes SC.

Definition 2 (defined update) *Assume two sets DU and SC which represent all possible data updates and schema changes, respectively. A change $c \in DU \cup SC$ is* ***defined on a given relation*** *R if one of the following conditions holds:*

- *if $c \in DU$, the schema of the tuple added or deleted must be equal to the schema of R.*
- *if $c \in SC$, the object c is applied to (an attribute or relation) must exist (for delete- and update-changes) or must not exist (for add-changes) in R.*

Definition 3 (valid update sequence) *A sequence of updates $(c_1, \ldots, c_n)$ with $c_i \in DU \cup SC$, denoted by ∂R, is called* ***valid for*** *R if for all i $(1 < i \leq n)$, c_i is defined on the relation $R^{(i-1)}$ that was obtained by applying $c_1, \ldots, c_{i-1}$ to R.*

For simplicity, we will also use the notation $\partial\omega$ to refer to a valid update sequence to the output table of an algebra operator ω. Note that these definitions naturally extend to views, since views can also be seen as relational schemas. For an example, consider propagation of the update `add-tuple('Berlin',1400,610) to LH` in Fig. 10 (p. 15). Having the value Berlin in the update tuple will lead to the addition of a new relation BERLIN in the output schema of the view—forming a sequence ∂V which contains both a schema change and a data update:

$$\partial V \quad = \quad (\textit{add-relation}(\mathsf{BERLIN}, (\mathsf{Type}, \mathsf{Destination}, \mathsf{BA}, \mathsf{LH})), \textit{add-tuple}(\mathsf{BERLIN}, (\text{'Economy'}, \mathsf{null}, 610)))$$

The *add-relation*-update is valid since the relation BERLIN did not exist in the output schema before, and the *add-tuple*-update is valid since its schema agrees with the schema of relation BERLIN defined by the previous update.

3.3 Overall Propagation Strategy

Given an *update sequence* implemented by a `List` data structure, our update propagation strategy works according to the algorithm in Fig. 4. Each node in the algebra tree has knowledge about the operator it represents. This operator is able to accept *one* input update and generate a sequence of updates as output. Each (leaf node) operator can also recognize whether it is affected by an update (by comparing the relation(s) on which the update is defined with its own input relation(s)). If it is not affected, it simply returns an empty update sequence.

After all the updates for the children of a node n are computed and collected in a list (variable s in the algorithm in Fig. 4), they are propagated one-by-one through n. Each output update generated by the operator of n when processing an input update will be placed into one update sequence, all of which are concatenated into the final return sequence r (see Fig. 4, $\leftarrow$ is the assignment operator).

```
function propagateUpdate(Node n, Update u)
    List r ← ∅,  s ← ∅
    if (n is leaf)
        if (n.operator is affected by u)
            r.append(n.operator.operatorPropagate(u))
    else
        for(all children c_i of n)
            /* s will change exactly once, see text */
            s.append(propagateUpdate(c_i, u))
        for(all updates u_i in s)
            r.append(n.operator.operatorPropagate(u_i))
    return r
```

Figure 4: The *SchemaSQL* View Maintenance Algorithm

The algorithm performs a postorder traversal of the algebra tree. This ensures that each operator processes input updates after all its children have already computed their output[2]. At each node n, an incoming update is translated into an output sequence ∂n of length greater than or equal to 0 which is then propagated to n's parent node. Since the algebra tree is connected and cycle-free (not considering joins of relations with themselves) all nodes will be visited exactly once. Also note that since updates occur only in one leaf at a time, only exactly one child of any node will have a non-empty update sequence to be propagated. That is, the first **for**-loop will find a non-empty addition to s only once per function call. After all nodes have been visited, the output of the algorithm will be an update sequence $\partial \mathrm{V}$ to the view V that we will prove to have an effect on V equivalent to recomputation.

3.4 Propagation of Updates through Individual *SchemaSQL* Operators

Since update propagation in our algorithm occurs at each operator in the algebra tree, we have to design a propagation strategy for each type of operator.

[2]We are not considering concurrent updates in this paper.

3.4.1 Propagation of Schema Changes through SQL Algebra Operators

The propagation of updates through standard SQL algebra nodes is simple. Deriving the update propagation for data updates is discussed in the literature on view maintenance [QW91, GL95]. It remains to define update propagation for selection, projection, and cross-product operators under schema changes [3]. In short, *delete-relation*-updates will make the output invalid, while other *relation*-updates do not affect the output. *Attribute*-updates are propagated by appropriate changes of update parameters or ignored if they do not affect the output. For example, a change *delete-attribute*(r, a) would not be propagated through a projection operator $\pi_{\bar{A}}$ if $a \notin \bar{A}$, and would be propagated as *delete-attribute*(q, a) otherwise, with q the name of the output relation of $\pi_{\bar{A}}$. We refer to our technical report [KR01] for further details, as they are not important for the comprehension of this paper.

3.4.2 *SchemaSQL* Operators

In Figs. 5–8, we give the update propagation tables for the four *SchemaSQL* operators. For the notation and meaning of variables and constants, please refer to Section 2.1. In order to avoid repetitions in the notation, the cases for each update type are to be read in an "if-else"-manner, i.e., the first case that matches a given update will be used for the update generation (and no other). Also, NULL-values are like other data values, except where stated otherwise.

Input Change	Conditions	Propagation
add-tuple (r, t)	$t[a_1, \ldots, a_n, a_p] \in R$	invalid view (key violation)
	$t[a_p] \in A^*$ $t[a_1, \ldots, a_n] \in R$	`update Q set` $[t[a_p]]$ `=` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_p] \in A^*$ $t[a_1, \ldots, a_n] \notin R$	`insert into Q` $(a_1, \ldots, a_n, a_p)$ `values` $(a_1, \ldots, a_n, a_d)$
	$t[a_p] \notin A^*$ $t[a_1, \ldots, a_n] \in R$	*add-attribute*$(q, t[a_p])$, `update Q set` $[t[a_p]]$ `=` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_p] \notin A^*$ $t[a_1, \ldots, a_n] \notin R$	*add-attribute*$(q, t[a_p])$, `insert into Q` $(a_1, \ldots, a_n, a_p)$ `values` $(a_1, \ldots, a_n, a_d)$
delete-tuple (r, t)	$t[a_p]$ exists in $R[a_p]$ exactly once	*delete-attribute*$(q, t[a_p])$
	$t[a_p]$ exists in $R[a_p]$ more than once	`update Q set` $[t[a_p]]$ `= NULL` `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ [4]
update-tuple (r, t, t')	$t[a_1, \ldots, a_n, a_p] = t'[a_1, \ldots, a_n, a_p]$	`update Q set` $[t[a_p]]$ `=` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_1, \ldots, a_n, a_p] \neq t'[a_1, \ldots, a_n, a_p]$	break down into (*delete-tuple*, *add-tuple*)
add-attribute(r, a)		*add-attribute*(q, a)
delete-attribute(r, a)	$a \in \{A_d, A_p\}$	invalid view
	$a \notin \{A_d, A_p\}$	*delete-attribute*(q, a)
rename-attribute(r, a, a')	$a = A_d$	$\text{UNFOLD}_{a_p,a} \Longrightarrow \text{UNFOLD}_{a_p,a'}$
	$a = A_p$	$\text{UNFOLD}_{a,a_d}(R) \Longrightarrow \text{UNFOLD}_{a',a_d}(R)$
	$a \notin \{A_d, A_p\}$	*rename-attribute*(q, a, a')
delete-relation(r)		*delete-relation*(q)
rename-relation(n, n')		$\text{UNFOLD}_{a_p,a_d}(N) \Longrightarrow \text{UNFOLD}_{a_p,a_d}(N')$ (renaming the input relation)

[4] if this update leads to a tuple with all NULL-values, the tuple must be deleted.

Figure 5: Propagation Rules for Q=$\text{UNFOLD}_{a_p,a_d}(R)$

[3] these are the only operators necessary for the types of queries discussed in this paper

Input Change	Conditions and Variable Binding	Propagation

add-tuple (r,t)	$(A^* = \{a_1^*, \ldots, a_k^*\}, k \leftarrow \lvert A^* \rvert)$	`for` i `:=` $1..k$
		`insert into Q`
		`values (`$a_1, \ldots, a_n$`,`a_i^*`,`$t[a_i^*]$`)`

delete-tuple (r,t)		`delete from Q` `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
update-tuple (r,t,t')	$A \in A^*$; set $t[a]$ to a value c	`update Q set` $a_d = c$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = a$
	$A \notin A^*$; set $t[a]$ from a value b to a value c	`update Q set` $a = c$ `where` $a = b$

add-attribute(r,a)	$A \in A^*$[5]	`foreach tuple` $u \in R$
		`insert into Q (`$a_1, \ldots, a_n, a_p, a_d$`)`
		`values (`$u[a_1, \ldots, a_n], a$`,NULL)`

	$A \notin A^*$	*add-attribute*(q,a)
delete-attribute(r,a)	$A \in A^*$	`delete from Q where` $a_p = a$
	$A \notin A^*$	*delete-attribute*(q,a)
rename-attribute(r,a,a')	$A \in A^*$	`update Q set` $a_p = a'$ `where` $a_p = a$
	$A \notin A^*$	*rename-attribute*(q,a,a')
delete-relation(r)		*delete-relation*(q)
rename-relation(n,n')		$\textsc{Fold}_{a_p,a_d}(N) \Longrightarrow \textsc{Fold}_{a_p,a_d}(N')$

[5]Note that the decision whether a *new* attribute should be a member of $a_1, \ldots, a_n$ can only be made by evaluating the view query.

Figure 6: Propagation Rules for Q=$\textsc{Fold}_{a_p,a_d,A^*}(R)$

Input Change	Conditions	Propagation

add-tuple (r,t)	$t[a_p] \notin A^*$	`add-relation [`$t[a_p]$`]`
		`with schema (`$S_R \backslash R.A_p$`);`
		`insert into [`$t[a_p]$`]`
		`values (`$t[a_1, \ldots, a_n]$`)`

	$t[a_p] \in A^*$	`insert into [`$t[a_p]$`]` `values (`$t[a_1, \ldots, a_n]$`)`
delete-tuple (r,t)	$t[a_p]$ exists in $R[a_p]$ exactly once	`delete-relation [`$t[a_p]$`]`
	$t[a_p]$ exists in $R[a_p]$ more than once	`delete from [`$t[a_p]$`]` `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
update-tuple (r,t,t')	$t[a_1, \ldots, a_n, a_p] = t'[a_1, \ldots, a_n, a_p]$	`update [`$t[a_p]$`] set [`a_d`] =` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_1, \ldots, a_n, a_p] \neq t'[a_1, \ldots, a_n, a_p]$	break down into (*delete-tuple*,*add-tuple*)
add-attribute(r,a)		$\forall q \in \{q_1 \ldots q_n\}$: *add-attribute*(q,a)
delete-attribute(r,a)	$a = A_p$	invalid view
	$a \neq A_p$	$\forall q \in \{q_1 \ldots q_n\}$: *delete-attribute*(q,a)[6]
rename-attribute(r,a,a')	$a = A_p$	$\textsc{Split}_a(R) \Longrightarrow \textsc{Split}_{a'}(R)$
	$a \neq A_p$	$\forall q \in \{q_1 \ldots q_n\}$: *rename-attribute*(q,a,a')
delete-relation(r)		$\forall q \in \{q_1 \ldots q_n\}$: *delete-relation*(q)
rename-relation(n,n')		$\textsc{Split}_{a_p}(N) \Longrightarrow \textsc{Split}_{a_p}(N')$

[6]If this update leads to a tuple with all NULL-values in an output relation, the tuple must be deleted.

Figure 7: Propagation Rules for Q=$\textsc{Split}_{a_p}(R)$

Input Change	Conditions and Variable Bindings	Propagation
add-tuple (r_x, t)		`insert into Q` $(a_1, \ldots, a_n, a_p)$ `values` $(t[a_1, \ldots, a_n], r_x)$
delete-tuple (r_x, t)		`delete from Q` `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = r_x$
update-tuple (r_x, t, t')	$A = A_d$; set $t[a]$ to a value c	`update Q set` $a = c$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = r_x$
	$A \neq A_d$; set $t[a]$ from a value b to a value c	`update Q set` $a = c$ `where` $a = b$ `and` $a_p = r_x$
add-attribute(r, a)	add simultaneously to all R_i	*add-attribute*(q, a)
	otherwise	invalid view
delete-attribute(r, a)	delete simultaneously from all R_i	*delete-attribute*(q, a)
	otherwise	invalid view
rename-attribute(r, a, a')	rename simultaneously in all R_i	*rename-attribute*(q, a, a')
	otherwise	invalid view
add-relation(r_x, S)		no change (until first *add-tuple* to R_x)
delete-relation(r_x)		`delete from Q where` $a_p = r_x$
rename-relation(n, n')		$\textsc{Unite}_{a_p}(\{R_1, \ldots, N, \ldots, R_n\}) \Longrightarrow \textsc{Unite}_{a_p}(\{R_1, \ldots, N', \ldots, R_n\})$

Figure 8: Propagation Rules for Q=$\textsc{Unite}_{a_p}(R_1, R_2, \ldots, R_n)$[7]

[7]Note that r_x is the name of Relation R_x, which is one of the n relations of equal schema that are united by the UNITE-operator.

Inspection of the update propagation tables shows several properties of our algorithm. For example, the view becomes invalid under some schema changes or data updates, mainly if an attribute or relation that was necessary to determine the output schema of the operator is deleted (e.g., when deleting the pivot or data attribute in UNFOLD). In the case of *rename*-schema changes (e.g., under *rename-relation* in FOLD), some operators change their parameters. Those are simple renames that do not affect operators otherwise. In those cases we denote renaming by $\Rightarrow$. The operator will produce a zero-element output sequence.

3.4.3 Formalization of the Propagation of Updates

A formalization of the propagation of updates is extensive and lacks the conciseness of the propagation tables given in this section. Therefore, we will only give an example of how such a definition could be accomplished. We will consider the propagation of *add-tuple* through UNFOLD (Fig. 9):

Using the notation from Section 2.2.3, assume a relation $R = (N_R, S_R, E_R)$ with n attributes that is the input for an operator $Q = \textsc{Unfold}_{a_p,a_d}(R)$ producing an output relation $Q = (N_Q, S_Q, E_Q)$ with $n - 2 + k$ attributes and an update to R, denoted by $\Delta_R = t[a_1, \ldots, a_n, a_p, a_d] = (x_1, \ldots x_n, x_p, x_d)$. Let A^* be a set of the k distinct values in A_p (the pivot attribute, see the definition of UNFOLD in Sec. 2.2).

The propagation of this update is shown in Fig. 9.

The structure $(E_Q \setminus T_1)^{\oplus}$ in the figure[8] is constructed by *adding an attribute* to $E_Q \setminus T_1$, i.e., $(E_Q \setminus T_1 \subseteq D_1 \times \ldots \times D_{n+k}) \Rightarrow ((E_Q \setminus T_1)^{\oplus} \subseteq D_1 \times \ldots \times D_{n+k} \times D_d)$ with all data values in this new attribute set to NULL ($\perp$). Note that the output relation becomes invalid iff an update is inserted into the input relation that agrees in $a_1, \ldots, a_n, a_p$ with an existing tuple (similar to a key violation).

[8]We use the symbol $\setminus$ to denote set-difference.

$$
\begin{array}{rcl}
E'_R & \leftarrow & E_R \cup \Delta_R \qquad \textit{a tuple-add} \\
 & \Downarrow & \\
\nu & \leftarrow & \Delta_R[a_p] \qquad \textit{the pivot-value of the new tuple} \\
T_0 & \leftarrow & \{t \in E_R \mid t[a_1, \ldots, a_n, a_p] = \Delta_R[a_1, \ldots, a_n, a_p]\} \qquad \textit{find out if added tuple exists in } E_R \\
S'_Q & \leftarrow & \begin{cases} \emptyset & \text{if } T_0 \neq \emptyset \qquad \textit{key violation} \\ S_Q & \text{if } \nu \in A^* \qquad \textit{note that } k = |A^*| \\ (a_1, \ldots, a_n, a_1^*, \ldots, a_k^*, a_p) & \text{otherwise} \qquad \textit{schema change if necessary} \end{cases} \\
T_1 & \leftarrow & \{t \in E_Q \mid t[a_1, \ldots, a_n] = \Delta_R[a_1, \ldots, a_n]\} \qquad \textit{the matching tuples in the output relation} \\
T_2 & \leftarrow & \{t \in T_1 \mid t[\nu] \leftarrow \Delta_R[a_d]\} \qquad \textit{set pivot attribute to value in data attribute} \\
T_3 & \leftarrow & \begin{cases} \{t[a_1, \ldots, a_n, a_1^*, \ldots, a_k^*, a_p] \mid t[a_1, \ldots, a_n, a_p] \leftarrow \Delta_R[a_1, \ldots, a_n, a_d];\ t[a_i^*] \leftarrow \perp\} \\ \hfill \text{if } T_1 = \emptyset \\ \hfill \textit{if new row in output table, construct new tuple, fill unused attributes with NULL} \\ \\ \{t[a_1, \ldots, a_n, a_1^*, \ldots, a_k^*, a_p] \mid t[a_1, \ldots, a_n, a_1^*, \ldots, a_k^*] \in T_1;\ t[a_p] \leftarrow \Delta_R[a_d]\} \\ \hfill \text{otherwise} \\ \hfill \textit{just set appropriate value} \end{cases} \\
E'_Q & \leftarrow & \begin{cases} \emptyset & \text{if } T_0 \neq \emptyset \\ (E_Q \setminus T_1) \cup T_2 & \text{if } \nu \in A^* \qquad \textit{no schema change} \\ (E_Q \setminus T_1)^{\oplus} \cup T_3 & \text{otherwise} \qquad \textit{an output schema change} \end{cases}
\end{array}
$$

Figure 9: Propagation of *add-tuple(Δ_R)* through an UNFOLD-Operator

3.5 Update Propagation Example

Fig. 10 gives an example for an update that is propagated through the *SchemaSQL*-algebra-tree in Fig. 2 (see also Fig. 15). All updates are computed by means of the propagation tables in the previous section.

The operators appear in boxes with their output attached below each box (SQL-statements according

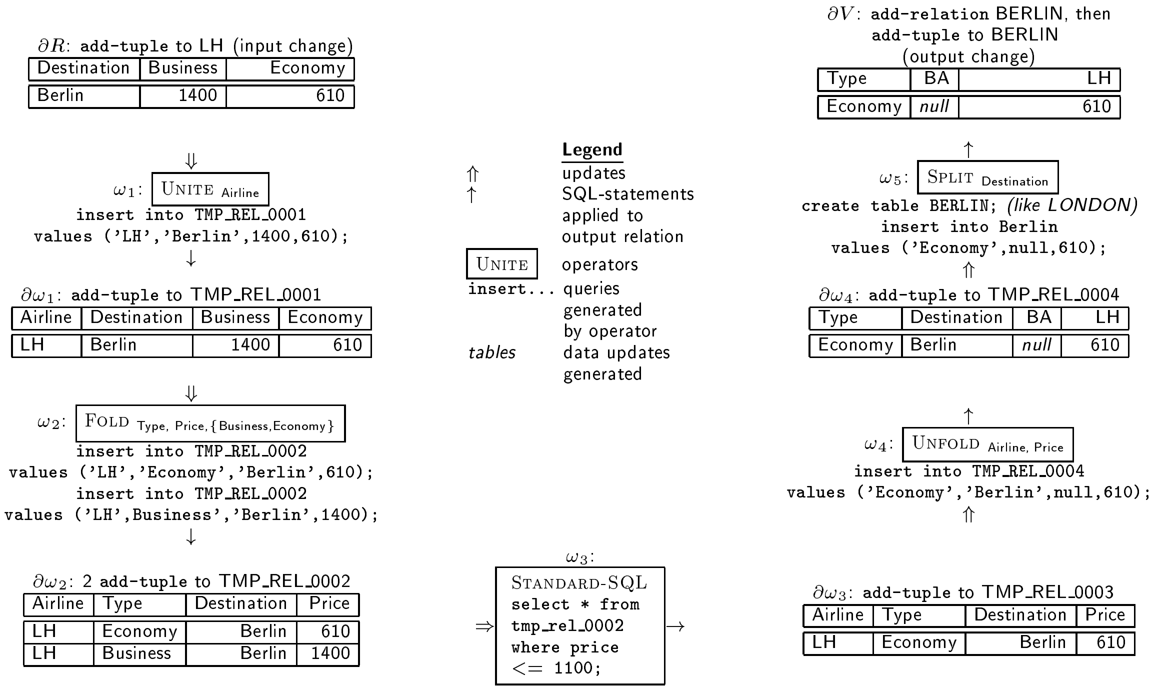

Figure 10: Update Propagation in the View from Figure 2. See Section 3.5 for explanation.

to our update tables in [KR01]). The actual tuples added by these SQL-statements are shown in tabular form. The sending of updates to another operator is denoted by double arrows ($\Uparrow$), while single arrows ($\uparrow$) symbolize the transformation of SQL-statements into updates. We are propagating an *add-tuple*-update to base relation LH. Algorithm *propagateUpdate* will perform a postorder tree traversal, i.e., process the deepest node (UNITE) first, and the root node (SPLIT) last. The operators are denoted by ω_1 through ω_5, in order of their processing. First, the UNITE operator propagates the incoming update into a one-element sequence $\partial\omega_1$ of updates which is then used as input to the FOLD-operator. The FOLD-operator propagates its input into a two-element sequence $\partial\omega_2$, sent to the StandardSQL-operator. This operator then propagates each of the two updates separately, creating two sequences $\partial\omega_{3_1}$ and $\partial\omega_{3_2}$, with 1 and 0 elements, respectively. Recall from Section 3.3 that in the case of more than one update sequence being created by an operator, those sequences can simply be concatenated before the next operator's propagation is executed, yielding $\partial\omega_3$. Since one update is not propagated due to the WHERE-condition in the StandardSQL-node, we have $\partial\omega_3 = \partial\omega_{3_1}$. UNFOLD now transforms its incoming one-element update sequence $\partial\omega_3$ into another one-element sequence $\partial\omega_4$ which becomes the input for the SPLIT-operator. This operator finally creates a two-element sequence, consisting of an *add-relation* schema change followed by an *add-tuple* data update. This sequence is the final update sequence ∂V which is applied to view V, leading to the new view V' equivalent to the view obtained by recomputation.

3.6 Grouping Similar *SchemaSQL* Updates in Batches

Certain updates in our strategy are transformed by some operators into update sequences ∂ in which all the updates are similar. This gives an opportunity for optimization on our update propagation strategy.

For example, a FOLD-node can transform a single schema change (such as a *attribute-delete*) into a sequence of data updates (such as a sequence of *tuple-deletes*). An inspection of the update propagation tables in this section shows that, typically, such a sequence consists of similar updates. Consider the example in Fig. 10, where a deletion of attribute Business in relation TMP_REL_0001 would lead to a sequence of *delete-tuple* updates of all tuples in TMP_REL_0002 that have the value Business in the attribute Type. A simple way of executing all those updates efficiently using SQL would be to issue a query such as `delete from TMP_REL_0002 where type='Business'`. Thus, instead of propagating all individual tuple updates using some delta relation, as done in traditional view maintenance, we instead propose to abstract this sequence of updates into an SQL update statement and push the complete statement through the algebra tree.

We have identified two classes of such *batched updates* that occur frequently as outputs of our propagation strategy, as described below.

Definition 4 (Batched Update) *A* batched update *is a sequence of* SchemaSQL *updates, denoted by ∂, which adheres to one of the following structures:*

- *∂ consists entirely of* delete-tuple-*updates to the same relation R, with equal schema and a set of attributes $a_1 \dots a_k$ whose values are a unique identifier for each tuple in ∂ (i.e., form a key). We denote such a sequence by*

  $$\text{delete-tuple-batch}(r, \mathit{cond}(a_1, c_1), \dots, \mathit{cond}(a_k, c_k))$$

  with $\mathit{cond}(a_i, c_i)$ a condition selecting tuples $t \in R$ that have value c_i in attribute a_i ($t[a_i] = c_i$). This represents a set of delete-tuple statements on the output relation R that could be generated by an SQL-`delete` statement with the WHERE-conditions $\mathtt{a_1} = \mathtt{c_1}, \dots, \mathtt{a_k} = \mathtt{c_k}$.

- *∂ consists entirely of* update-tuple-*updates to relation R. All update-tuples have equal schema. Parameters are a single attribute b with (unique or duplicate) values, and a function f between the old and new values of another attribute a in each tuple. We denote such a sequence by*

  $$\text{update-tuple-batch}(r, a, f, b, c)$$

  with a, b denoting attribute names, f denoting a function over the domain of the attribute with name a (that is, $f : D_a \to D_a$), and c denoting a constant. This represents a set of update-tuple updates affecting every tuple t for which the value of attribute b is c, by changing the value $t[a]$ to $f(t[a])$, i.e., $\forall t \in R$ s.t. $t[b] = c : \; t[a] \leftarrow f(t[a])$. In words, the update update-tuple-batch(r, a, f, b, c) *means "in relation r, set $a = f(a)$ where $b = c$". Note that for simplicity, we are restricting batched updates to a single WHERE-condition.*

With this definition of batched update, the above example can now be represented as *delete-tuple-batch* (TMP_REL_0002, *cond*(type, 'Business')).

We do not define insert-tuple batches since we consider only single data updates or schema changes entering our algebra tree, and such updates will never be transformed into larger "batches". In particular, adding an attribute or a relation in a base table means adding an *empty* structure containing no data. As only structures with matching schemas (i.e., attribute of a matching data type or relations with a matching set of attributes) can be added to the information space, the only new information to the system is the *name* of the new attribute or relation, respectively. Thus, such updates do not lead to batches of updates, and in fact often do not lead to any updates on the view extent at all.

Batches of schema-changes are also not useful because meaningful schema-change batches do not occur in our context. Inspection of the update tables in this section shows that, with the exception of the SPLIT node, propagation of schema changes always leads to a *single* schema change, not sequences of related changes. In the case of the SPLIT-node, any resulting "batch" of schema changes will lead to changes across several relations, an operation that cannot be optimized using our batched-approach and SQL-statements.

As mentioned above, the main benefit of batched updates lies in a possible optimization of the implementation of our update propagation strategy. Since some operators generate batches of related updates, considering batches as types of updates and propagating those through the algebra tree just like single updates could lead to performance improvements of the system. For an example, consider again Fig. 10 and an update *delete-attribute*(TMP_REL_0001, Business) as input to the FOLD-operator. Setting $n = |\mathsf{TMP\_REL\_0001}|$, this update in the current strategy would lead to a propagation of n single *delete-tuple* updates, whereas a treatment of all those updates as a batch would require the propagation of only one update, namely *delete-tuple-batch*(TMP_REL_0002,*cond*(type,'Business')). Figures 11–14 show the propagation tables. As before, the input table is denoted by R (with name r) and the output table by Q (with name q). The remaining syntax follows Def. 4.

Input Change	Parameters	Conditions	Propagation
delete-tuple-batch	$(r, cond(a, c))$	$a = a_p$	*delete-attribute(q,c)*

		$a = a_d$,
		A^* unchanged
			`foreach` $a \in A^*$
			`update Q`
			`set` $a =$ `NULL`
			`where` $a = c$

		other	*delete-tuple-batch* $(q, cond(a, c))$
	$(r, cond(a_1, c_1), \ldots,$ $cond(a_n, c_n))$		*delete-tuple-batch* $(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$
update-tuple-batch	(r, a, f, b, c)	$f(v) = c_{\text{new}}$ (a constant function), $a = b = a_p$	*rename-attribute(q,c,*c_{new}*)*
		$a = a_d, b = a_p$	`update Q set` $[c] = f([c])$ or *update-tuple-batch* $(q, c, f, null, null)$
		$a = a_d, b \neq a_p$	`foreach` $\nu \in A^*$ *update-tuple-batch*(q, ν, f, b, c)

Figure 11: Batched Update Propagation Rules for Q=UNFOLD $_{a_p,a_d}(R)$

4 Correctness

Our update propagation strategy is equivalent to a stepwise evaluation of the algebraic expression constructed for a query. Each operator transforms its input changes into a set of semantically equivalent output changes, eventually leading to a set of changes that must be applied to the view to synchronize it with the base

Input Change	Parameters	Conditions	Propagation
delete-tuple-batch	$(r, cond(a, c))$	$a \in A^*$	delete from Q where $a_p = a$ and $a_d = c$
		other	*delete-tuple-batch* $(q, cond(a, c))$
	$(r, cond(a_1, c_1), \ldots,$ $cond(a_n, c_n))$		*delete-tuple-batch* $(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$
update-tuple-batch	(r, a, f, b, c)	$a \in A^*, b \in A^*$	update Q set $a_d = f(a_d)$ where $a_p = a$ and $b = c$
		other	*update-tuple-batch*(q, a, f, b, c)

Figure 12: Batched Update Propagation Rules for Q=FOLD $_{a_p, a_d, A^*}(R)$

Input Change	Parameters	Addtl. Conditions	Propagation
delete-tuple-batch	$(r, cond(a, c))$	$a = a_p$	*del-relation(q,c)*
		$a = a_d$, A^* unchanged	foreach $q \in A^*$ update q set $q.a_d = \mathsf{NULL}$ where $q.a_d = c$
		other	foreach $q \in A^*$ *delete-tuple-batch*$(q, cond(a, c))$
	$(r, cond(a_1, c_1), \ldots,$ $cond(a_n, c_n))$		foreach $q \in A^*$ *delete-tuple-batch* $(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$
update-tuple-batch	$(r, a_p, f, b, c_{\text{old}})$ with $f(v) = c_{\text{new}}$ (a constant function)	$b = a_p$	*rename-relation*$(c_{\text{old}}, c_{\text{new}})$
		$b \neq a_p,\ a \in A^*$	foreach $q \in A^*$ *update-tuple-batch*(q, a, f, b, c)

Figure 13: Batched Update Propagation Rules for Q=SPLIT $_{a_p}(R)$. Note that $A^* = \{q_1, q_2, \ldots, q_k\}$ is the set of output relation names.

Input Change	Parameters	Addtl. Conditions	Propagation
delete-tuple-batch	$(r_x, cond(a, c))$		delete from Q where $a_p = r_x$ and $a = c$
update-tuple-batch	(r_x, a, f, b, c)	$a \in A^*$	update Q set $a = f(a)$ where $a_p = r_x$ and $b = c$

Figure 14: Batched Update Propagation Rules forQ=UNITE $_{a_p}(R_1, R_2, \ldots, R_n)$

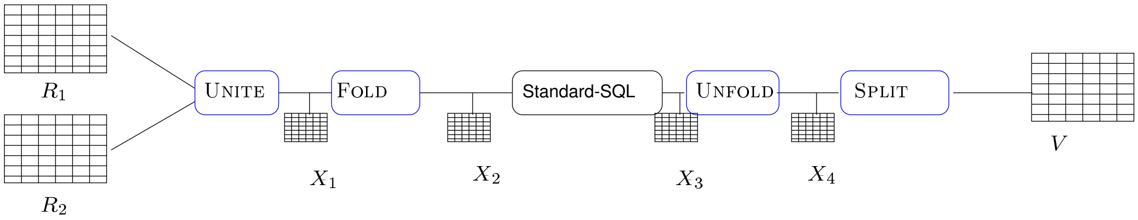

Figure 15: A *SchemaSQL* Algebra Tree.

relation change. In this section, we will show that this strategy leads to correct update propagation. Recall that we denote an update sequence applied to relation R by ∂R. We will use the notation R for the input relation and Q for the output relation throughout this section.

Before we prove the correctness of the algorithm, we state some observations: The structure of the algebra tree for a view depends only on the query, not on the base data [LSS99]. The only changes to operators under base relation updates are possible changes of parameters (schema element names) inside the operators; an algebra operator can not disappear or appear as the result of a base update. However, the entire view query may be rendered invalid, for example under some *delete-relation*-updates.

Furthermore, an inspection of the update propagation algorithm (Fig. 4) shows that the propagation of any single base relation update occurs strictly along a path in the algebra tree, strictly from a leaf to the root. That is, only *SchemaSQL* algebra operators along the single path from the updated base relation to the root are affected by an update. This is in contrast to SQL view maintenance, where maintenance queries to related sources are necessary for some operators. The four *SchemaSQL* operators do not combine input relations in a way similar to an SQL-join, so that maintenance queries to other branches of the algebra tree are not generated by the *SchemaSQL* operators. For correctness of standard SQL maintenance queries (which only occur for some operators such as *join*), we rely on well-known related work.

Let us label the output relations of each operator along the path of update propagation with $X_1, \ldots, X_n$, in ascending order from the operator closest to the leaf to the operator closest to the root of the algebra tree. In Fig. 15, we have labeled the output relations of each operator $(X_1, \ldots, X_4)$, as well as the base relations (R_1, R_2) and the view (V).

We first prove correctness of operators and then show the overall propagation scheme to be correct.

Theorem 1 (Correctness of Incremental Propagation for Individual Operators) *Let* $\omega \in \{\textsc{Unite}, \textsc{Split}, \textsc{Unfold}, \textsc{Fold}, \pi, \sigma, \times\}$ *be a node in a* SchemaSQL *algebra tree. Let R be the input relation(s) for ω and $Q = \omega(R)$ be its output relation(s). Furthermore, let ΔR be a data update or schema change to R, transforming $R \to R'$ and Q_{REC} the output relation of ω after recomputation. Applying the rules from the update propagation tables Figs. 5–8 and Section 3.4.1 for ω and ΔR will generate a sequence of updates defined on the node's output relation (denoted by ∂Q, see Def. 2) that transforms $Q \to Q_{\mathsf{INC}}$, with $Q_{\mathsf{INC}} = Q_{\mathsf{REC}}$.*

Proof: The proof is given by inspecting the update propagation tables, Figs. 5–8, and comparing their output with the expected output after recomputation for each case. Due to space constraints, we can only give two examples for such comparisons as a proof idea. Consider the propagation of a *delete-tuple* data update in the FOLD-operator (Fig. 6). Let a relation R be folded by $Q = \textsc{Fold}_{a_p,a_d,A^*}(R)$. Now consider the relation $R' = R \backslash \{t\}$, with tuple t deleted. Note that t has up to $|A^*|$ non-null values in its data attributes (i.e., in attributes whose names are in A^*). For each of those non-null values, the pre-update output relation Q contained a separate tuple which now has to be deleted. Therefore, after recomputation, the FOLD-operator produces an output relation Q' that differs from Q in that it has up to $|A^*|$ tuples less. All those missing tuples have as a common feature that they agree in the values of their attributes $a_1, \ldots, a_n$ (i.e., all attributes except the ones in A^*) with the deleted tuple. This is precisely what the update propagation rule (line 2 of Fig. 6) accomplishes by deleting all tuples with that condition.

Let us also consider the propagation of the *delete-attribute* schema change in FOLD (line 5 of Fig. 6). Recomputation of the operator yields a Q' that differs from Q in one of two ways: if a data attribute A ($a \in A^*$) in R is deleted, all tuples whose values in A_p correspond to the name of A are missing from Q'. If a non-data attribute is deleted from R, the attribute in Q' that has the same name as the deleted attribute in R is deleted. In both cases, the update propagation rules change Q in exactly that way.

The remaining operators and cases can be verified in a similar fashion. □

The following corollary is immediate since if an update sequence correctly transforms a relation, it must also be valid (Def. 3) on that relation.

Corollary 1 *The propagation of any update defined on input relation R through an operator ω will produce a valid update sequence for output relation Q.*

Theorem 2 (Correctness of *SchemaSQL* View Maintenance) *Let V be a view defined over the set of base relations $R_1, \ldots, R_p$, and $\Delta R_u \in \{DU, SC\}$ an update applied to one relation R_u $(1 \leq u \leq p)$. Let R'_u be the relation R_u after the application of ΔR_u and V'_{REC} be the view after recomputation. Furthermore, let the* SchemaSQL *View Maintenance Algorithm as defined in Section 3.3 produce a change sequence ∂V that transforms view V into view V'_{INC}. Then, $V'_{\mathsf{REC}} = V'_{\mathsf{INC}}$.*

Proof: Let n be the number of intermediate relations X_i affected by an update (along the path from R_u to V). We want to prove that recomputation generates the same intermediate relations (and therefore the same view relation) as incremental updating, i.e, $\forall i \; (1 \leq i \leq n) : (X'_i)_{\mathsf{REC}} = (X'_i)_{\mathsf{INC}}$ and thus $V'_{\mathsf{REC}} = V'_{\mathsf{INC}}$. The proof is by induction over X_i for $i = 0 \ldots n+1$. Set $X_0 = R_u$, $(X'_{n+1})_{\mathsf{REC}} = V'_{\mathsf{REC}}$, $(X'_{n+1})_{\mathsf{INC}} = V'_{\mathsf{INC}}$.

Base Case: The base case for $i = 0$ is trivial. R'_u is the same relation, whether the algebra tree is recomputed or incrementally updated, i.e., $(X'_0)_{\mathsf{REC}} = (X'_0)_{\mathsf{INC}} = R'_u$.

Induction Hypothesis: $(X'_k)_{\mathsf{REC}} = (X'_k)_{\mathsf{INC}}$ $(k \geq 0)$.

Induction Step: It is to show that $(X'_{k+1})_{\mathsf{REC}} = (X'_{k+1})_{\mathsf{INC}}$.

Since by hypothesis, $(X'_k)_{\mathsf{REC}} = (X'_k)_{\mathsf{INC}}$, there must exist an update sequence ∂X_k that correctly transforms X_k to X'_k (and must therefore be valid on X_k). Let us denote the operator whose input table is X_k by ω_k and let $m = |\partial X_k|$. By Thm. 1, any single valid update to any relation is correctly propagated through any one operator, in the sense that recomputation of the operator will yield the same result as incremental propagation. If $m = 1$, the induction step is thus proven. For $m > 1$, a valid sequence of m updates on X_k will trigger a sequence of incremental propagation steps in ω_k. This will cause ω_k to transform X_{k+1} into a sequence of m intermediate (temporary) relations $(X^{(1)}_{k+1})_{\mathsf{INC}} \ldots (X^{(m)}_{k+1})_{\mathsf{INC}}$, each of which is equivalent to the corresponding state $(X^{(1)}_{k+1})_{\mathsf{REC}}, \ldots, (X^{(m)}_{k+1})_{\mathsf{REC}}$ that could be reached by recomputing ω_k after each update. Note that $X^{(m)}_{k+1} \equiv X'_{k+1}$. After application of all m updates to X_{k+1} we have $X'_{k+1} = (X^{(m)}_{k+1})_{\mathsf{INC}} = (X^{(m)}_{k+1})_{\mathsf{REC}}$, or $(X'_{k+1})_{\mathsf{INC}} = (X'_{k+1})_{\mathsf{REC}}$. If any valid sequence of updates gets propagated correctly, the sequence ∂X_k (valid by Corollary 1) in particular must also be correctly propagated, i.e, produce a relation $(X'_{k+1})_{\mathsf{INC}}$ with $(X'_{k+1})_{\mathsf{INC}} = (X'_{k+1})_{\mathsf{REC}}$. **q.e.d.** □

5 Implementation

5.1 *SchemaSQL* Query Engine

The update propagation strategy described in this paper has been implemented in Java on top of a *SchemaSQL* query evaluation module also written by us. This query engine was built along the lines of [LSS99]. Our code first parses *SchemaSQL* queries (using JavaCC), then builds an algebra tree out of the parsed query, and finally evaluates the query result through a postorder traversal of that tree, computing the output of each algebra node as it is visited (cf. Fig. 16). The next node then reads its child node's temporary relation to compute its output. For this prototype implementation, each node temporarily stores its output through JDBC in the query engine's "local" relational database (Oracle 8) as keeping relations in memory only incurs limitations on the size of input relations and also would have required us to reimplement significant parts of relational query technology. For performance reasons, all intermediate nodes in the algebra tree share one JDBC-connection to the local database.

The implementation of the query engine uses pure Java and JDBC-connections to several instances of Oracle 8. We use standard SQL DDL and DML statements (`select`, `insert`, `delete`, `update` and statements for schema change operations like `alter table`) for all queries—thus making full use of the source database's SQL query evaluation capabilities. We do not use any system specific functions other than simple schema changes. A wrapper class (which we have also successfully implemented for Microsoft's Access) makes differences in the syntax of schema change operations transparent. Therefore, the implementation is independent of the database used.

To improve performance and simplify our code, our implementation attempts to use as much of the SQL query capability as possible, in particular by extracting standard SQL out of the *SchemaSQL* query and evaluating it in a single `StandardSQL`-node, which simply executes its stored SQL-statement against the local SQL-database. The *SchemaSQL* query engine currently does not perform or utilize any query optimization strategies other than those provided by the underlying SQL query engine when executing queries against the local database.

5.2 Incremental Update Propagation

To perform update propagation in the manner described in this paper, we added update propagation capabilities to each algebra operator class (UNITE, FOLD, UNFOLD, SPLIT, StandardSQL). A method `propagateUpdate()` in each node accepts one update and returns a list of (data and/or schema) updates which represent the result of the update propagation. Then, we added code for the propagation of the output updates of each operator to its parent. Thus, the same code that performs the postorder traversal of the operator tree for the initial materialization of the view can now also perform the incremental update propagation, by simply calling the update propagation method instead of the materialization method on each node and recursively using each operator's output as the input for the parent operator.

Updates were modeled into a small class hierarchy consisting mainly of the classes `SchemaUpdate` and `DataUpdate`. Each node in the algebra tree is now extended by the ability to propagate all such updates. The current code does not support batched updates.

Fig. 16 shows the architecture of our system.

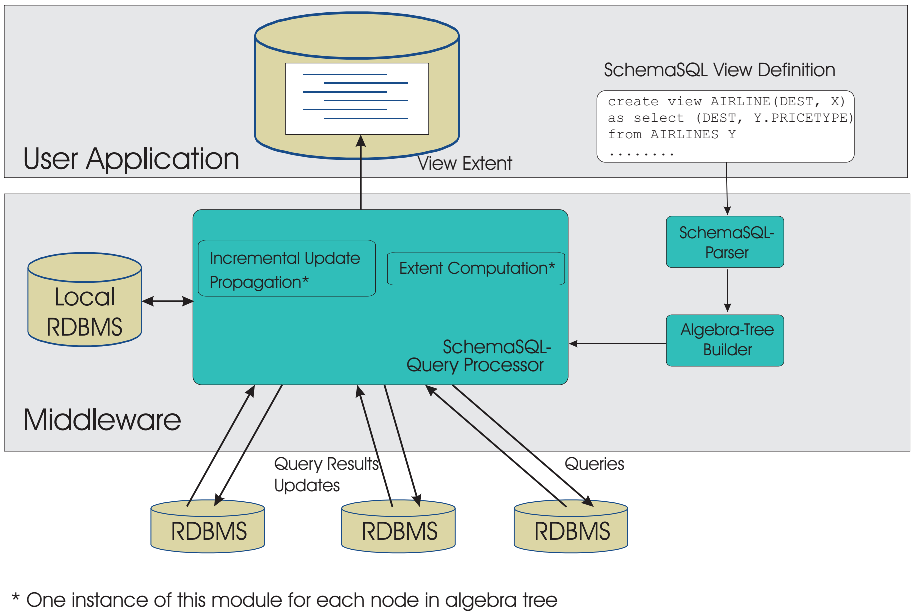

Figure 16: The Architecture of the *SchemaSQL* View Maintenance System

6 Performance Evaluation

6.1 Experimental Setup

6.1.1 Performance-Relevant Factors

As explained in this paper, *SchemaSQL* update propagation is significantly different from traditional incremental view maintenance. Major differences are the transformation of data updates into schema changes and vice-versa, the need for the propagation of base schema changes, and the propagation strategy based on propagating an update through an algebra tree rather than computing delta-queries against the base relations.

To assess the influence of those issues on performance, we executed a number of experiments on our prototype. We are reporting some of the results in this section. For the experiments described here, we focused on the following factors contributing to *SchemaSQL* update propagation performance:

- the type of update (data update or schema change) at the base relations;
- the transformation type of the update (i.e., the type, data update or schema change, into which a base update is propagated);
- the selectivity of conditions in the view query that determines the size of the view relative to the sizes of the base relations;
- the size of base relations.

6.1.2 Schema and View Queries

If not stated otherwise, all our experiments use the following view query, over the same base schema as in our running example (Fig. 1):

```
create view CITY(Type, AIRLINE) AS
  select PRICETYPE, FLIGHT.PRICETYPE
  from   -> AIRLINE,
         AIRLINE FLIGHT,
         AIRLINE-> PRICETYPE,
         FLIGHT.Destination CITY
  where PRICETYPE <> 'Destination'
  and AIRLINE like 'AIRLINE%'
  and FLIGHT.PRICETYPE <= '1101';
```

The output schema of this query, considering the input schemas from Fig. 1, consists of two relations Business and Economy which both have the schema (Destination,AIRLINE_1,...,AIRLINE_K), with one attribute named AIRLINE_X for each relation named AIRLINE_X in the input schema. The output schema may change during an experiment.

The base data was generated from a list of strings (representing city names), augmented by random numbers representing "flight prices". Those numbers were generated using uniformly distributed random numbers in a certain range. The base relation sizes and distribution of updates are described with each experiment.

Since we have multiple output relations, we need to extend the concept of view size to multiple relations. We thus define the *view size* to be the *sum* of the sizes of all output relations.

6.1.3 Measurements and System Parameters

The test system was a Pentium II/400 running Linux and Java 1.2.2. Database connectivity was achieved through JDBC. The databases used for our tests (local database and information source) were two installations of Oracle 8i, running on a Pentium 233 under Windows NT and a 4-processor 300MHz DEC Alpha under DEC OSF1, respectively.

For our experiments, we measured the total execution time of the initial materialization of the view (for control purposes, not shown in the chart), then the time for a number of updates, depending on the experiment, and finally the time for a recomputation of the view extent. The times were measured by comparing the system time before and after executing update propagation or recomputation, i.e., they include system, user, and I/O time.

6.2 Deleting Base Relations of Different Sizes

In our classification of updates (Section 3), inserting a table implies inserting an *empty* table. The data would have to be added in subsequent data updates. Therefore, inserting (and also renaming) schema elements leads to relatively simple propagation results, as the information content of the database does not change much under such updates. Therefore the experiments reported here concentrate mainly on the deletion of schema elements (attributes, relations), as well as on the insertion and deletion of data. We will report on view maintenance time both with and without the optimization of batched updates (Sec. 3.6).

We ran the above query over a schema containing four base relations (representing four different airlines) with approximately 100, 200, 300, and 400 tuples, respectively. We then deleted each of those base relations and compared the time for incremental update propagation with the time for recomputation after a base relation was deleted. After each deletion and measurement, the original information space was restored. With the above query, the original view extent had two relations Business and Economy with 378 and 444 tuples, respectively and decreased roughly proportionally after deletions of base relations. Fig. 17 shows the times measured.

Without the batched-update optimization, deleting a base relation R in a query like above will result in the creation of approximately $|R|$ updates inside the operator tree. Therefore, deleting larger relations takes longer than deleting smaller relations. On the other hand, recomputation time will decrease with larger sizes of the deleted relation, as the resulting view extent has less tuples. The crossover point between view maintenance and recomputation is at about a base size of 225, i.e., deleting a relation of more than 225 tuples (about 25% of the view size) will take longer than recomputing the view. This means that our update propagation strategy will perform better than recomputation for tuple-wise deletions of up to 25% of the entire information space. Propagating 225 delete-tuple updates will have a smaller total execution time than a single recomputation of the view. On the other hand, this experiment shows that a single input update (in this case a *delete-relation* update) can be very expensive as it may lead to many updates in the view extent. On the other hand, when using *batched updates*, the deletion of a base relation translates into a batched update, which is then propagated as far as possible as a batched update as well. In the case of this particular query, the deletion of a relation can be propagated through the UNITE, FOLD, and StandardSQL nodes as a *delete-tuple-batch*, and is then turned into a *delete-attribute* update in the UNFOLD-node (cf. the propagation table in Sec. 3.6). This explains the very low update propagation time for the delete-relation schema change in this case.

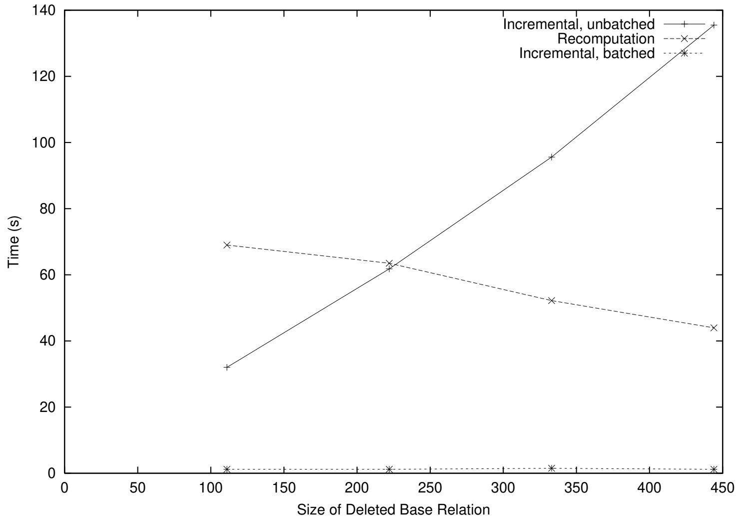

Figure 17: View Maintenance and Recomputation Times vs. Size of Deleted Base Relations

We also ran the same set of updates under the view query from our running example (Fig. 1), which creates one output relation for each unique destination in any of the input relations. Given that the input tables contain a large number of unique destinations (cities), many output tables are generated, at worst one output table per tuple in each base relation.

Note that in this case, the deletion of a base relation eventually leads to *one delete-attribute schema change per output relation*, so deleting a base relation with 200 tuples will first lead to the propagation of 200 *delete-tuple* updates, propagated to one output relation each, and then to many *delete-attribute* schema changes—exactly one for each output relation. The experiment shows that in this case, which incurs many schema changes in the output relations, a recomputation has a performance advantage over the unoptimized incremental update propagation. However, with batched updates, significant performance benefits can be realized that make incremental view maintenance almost as fast as recomputation. Figure 18 shows the results of this experiment. The incremental view maintenance still takes significant time since many schema changes still have to be executed on the output relations of the view.

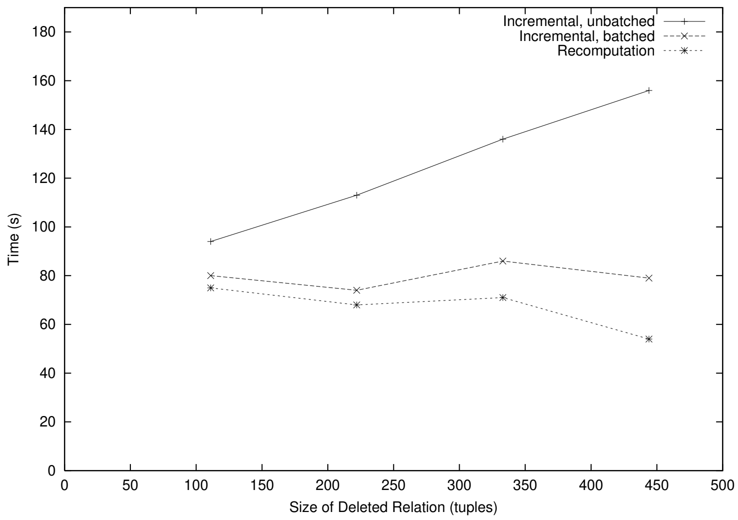

Figure 18: View Maintenance and Recomputation Times for a View with Many Relations

6.3 Deleting Tuples from Base Relations

In this experiment (Fig. 19), we delete a sequence of random tuples from the base schema and measured the cumulative propagation time. For our chart, we numbered the updates by consecutive numbers i. The cumulative propagation time for update i is the sum of the propagation times for all updates numbered $0 \ldots i$. We also measured the time to recompute the view after the entire update sequence was executed. This experiment shows again that in our schema, the crossover point between incremental maintenance and recomputation is at roughly 200 tuples, i.e., after 200 updates, recomputation of our view (with view size 888) would become more efficient. The view size is the major factor determining the recomputation time for a view, whereas the time for incremental propagation of a single data-update mainly depends on the system implementation, i.e., is roughly constant for our implementation and test environment. The average time to propagate a single update can be estimated from the slope of the curve in Fig. 19 to be about 285ms, which is a value depending mainly on the size of the tuples propagated.

From those facts, it can be concluded that the *ratio* between the number of propagated updates at the crossover point and the view size is a system constant depending only on the implementation, i.e., we expect that recomputation of a view will take roughly the same time as the incremental propagation of the deletion of a certain percentage of the view's tuples. In our experiment, this ratio was about 1/4 (a crossover point of 200 for a view with roughly 800 tuples).

Note the jump in the incremental maintenance time at the end of the curve. This time represents a data update that led to a schema change in the output relation. The reason is that the last tuple from a base relation was deleted which led to a *delete-attribute* change in an output relation.

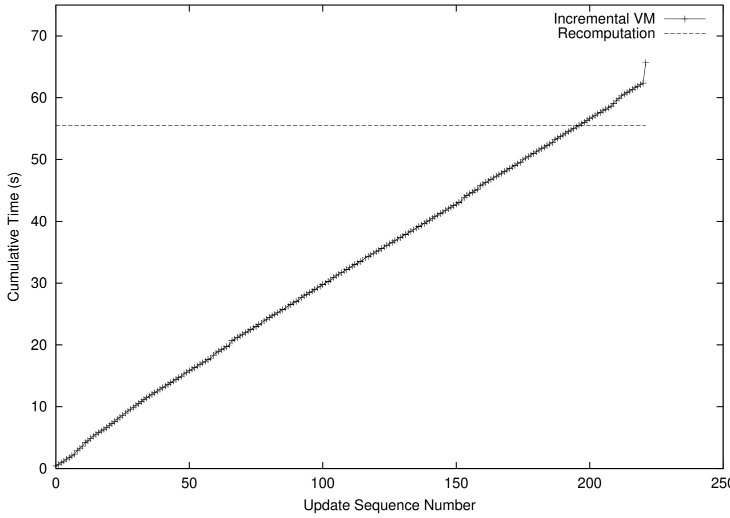

Figure 19: Deleting Tuples from the Input Schema

6.4 Deleting Tuples Leading to Schema Changes

In this experiment, we wanted to assess the difference in propagation time of the same updates, depending on whether these updates lead to data updates or schema changes in the view. Schema changes actually executed on a relation database are slow operations. Therefore, the expectation is that an update propagation (including the application of those updates against a database) that leads to a schema change will be slower than an update propagation leading to only a data update in the view.

Thus, in this experiment we are deleting tuples from the base schema. This time we have four base relations R_1, R_2, R_3, R_4 of sizes 1, 10, 100, and 1000, respectively, but make sure that some of the updates incur schema changes in the output schema. First, we inserted 10 tuples into relation R_1, then removed tuples from relation R_2 one-by-one, until R_2 was empty. This leads to a schema change in the output schema since the corresponding attribute is removed from each output relation. Then, we removed all 11 tuples from relation R_1 (incurring another schema change), followed by a removal of 10 random tuples from relation R_3. Note that in this sequence, updates #19 and #30 lead to schema changes in the output.

We then plotted the time each update took to propagate. Note the relatively even distribution of update propagation times around 250 ms in Fig. 20, except for two updates, which take over 2 seconds to propagate. Those are clearly those updates that led to schema changes in the output relations. The measured update time (if no schema change is incurred) is similar to the average time measured in Fig. 19. Again, the propagation time for the data updates depends mainly on the tuple size, and the underlying database, whereas the time for schema changes depends on the underlying database only (as practically no data has to be transported).

6.5 View Selectivity

In this experiment, we measure how the performance advantage of incremental view maintenance over recomputation is affected by the view selectivity, i.e., by the probability that a base tuple's data will actually be reflected in a view.

To assess the effect of different view selectivities on view maintenance times, we ran an experiment

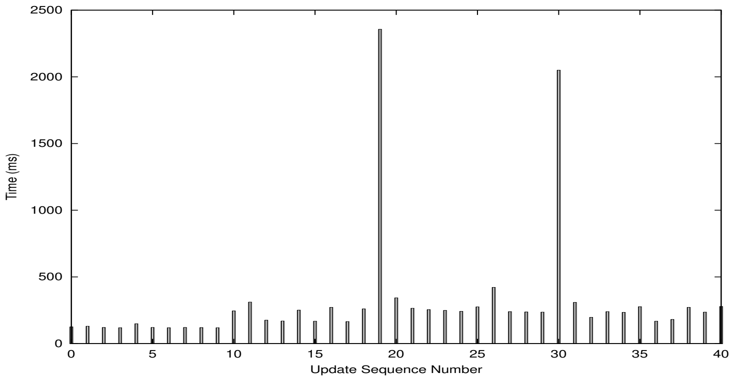

Figure 20: Base Updates lead to Data Updates or Schema Changes

over different selectivities in the view query. We adjusted selectivity in the range $[0.02 \ldots 1]$ by using different constant values for local conditions in the WHERE-clause of our query (i.e., conditions of the type `FLIGHT.PRICETYPE<=1100`). We define view selectivity over our multiple-relation output schema in analogy to view size as the ratio of the view size of the current query and the view size of a query without WHERE-clause. For each selectivity setting, we deleted a relation with 100 tuples (10% of the input tuples) from the base schema and measured incremental view maintenance time and view recomputation time. Fig. 21 shows the result of the experiment. The graph shows that both incremental view maintenance time and recomputation time increase with the view selectivity, which is not surprising, since in both cases more tuples have to be processed when the view selectivity (and thus the view) becomes larger. However, the relative increase in the incremental update propagation time is similar to the relative increase in recomputation time, meaning that our propagation strategy will keep its performance benefits under changes of the view's selectivity.

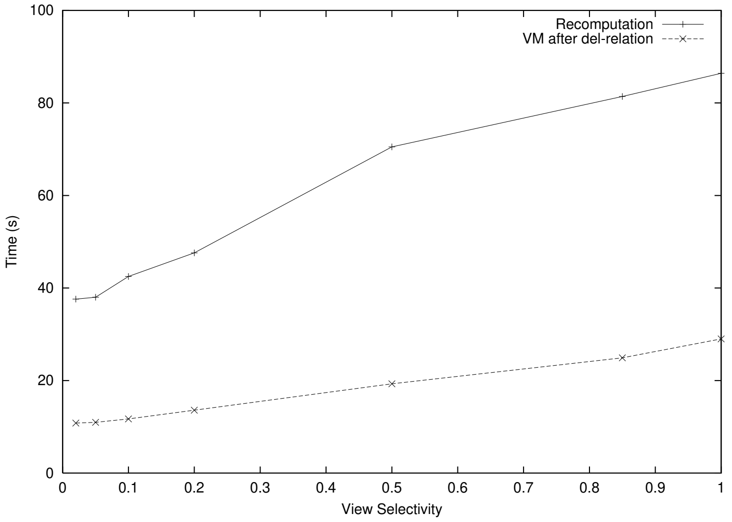

Figure 21: Update Propagation under Views of Different Selectivities

7 Related Work

The integration of data stored in heterogeneous schemas has long been an object of intensive studies. The problem of schematic heterogeneity or different source capabilities is repeatedly encountered when attempting to integrate data. Some examples are Garlic [TAH+96], TSIMMIS [HGMI+95], and DISCO [TRV95]. Several logic-based languages have been developed to integrate heterogeneous data sources (e.g., work by Krishnamurthy et al. [KLK91], HiLog [CKW89] or SchemaLog [GLS+97]). Some SQL-extensions have also been proposed, such as MSQL [LAZ+89] which has capabilities for basic querying of schema elements, and, in particular, *SchemaSQL* ([LSS96, LSS01], see below).

Those approaches are able to overcome different classes of schematic heterogeneities. However, the important class of schematic heterogeneities in semantically equivalent relational databases is often excluded from integration language proposals. Miller et al. [MIR93, Mil98] show that relational databases may contain equivalent information in different schemas and give a formal model (Schema Intension Graphs) to determine such "semantic equivalence" of heterogeneous schemas.

The predominant approach at integrating such semantically equivalent schemas has been done by Gyssens et al. [GLS96] and later by Lakshmanan, Sadri, and Subramanian [LSS96, LSS99]. In [LSS96], the authors present *SchemaSQL*, which is used as the basis for our work. A more thorough treatment of the language and, in particular, its use for aggregation, is given in [LSS01]. *SchemaSQL* builds upon earlier work in SchemaLog [GLS+97]. It is a direct extension of SQL, with the added capability of querying and restructuring not only data, but also schema in relational databases, and transforming data into schema and vice-versa. Thus, using *SchemaSQL* as a query language makes it possible to overcome syntactic (schematic) heterogeneities between relational data sources.

A second foundation of our work is the large body of work on incremental view maintenance. After early models for view maintenance [BLT86], many algorithms for efficient and correct view maintenance under a variety of assumptions have been proposed. Prominent results, often taking concurrency into account, include ECA [ZGMHW95], SWEEP [AESY97], Mohania et al. [MKK97] and the approach by Gupta et al. [GMS93]. Those approaches follow an *algorithmic* approach in that they propose algorithms to compute changes to a view.

Griffin and Libkin [GL95] consider views with duplicates, and, more importantly, follow an *algebraic* approach which defines a complete and minimal set of relational algebra operators. They achieve a rigorous proof of the correctness of view maintenance by proving the correctness of those operators and their nesting. This is an idea that we have adopted for our work. Griffin and Libkin's work is partly based on the algebraic approach by Qian and Wiederhold [QW91]. Colby et al. [CGL+96] correct the *state bug* that occured in earlier work by Griffin and Libkin as well as in other authors' proposals. Our work builds on existing incremental view maintenance literature but handles schema changes in addition to data updates. Since *SchemaSQL* is able to transform data updates into schema updates and vice-versa, any operator in the *SchemaSQL* algebra tree must be able to propagate both classes of updates. To our knowledge, our view maintenance strategy is the first that achieves incremental update propagation in such schema-restructuring views.

Another body of literature that is related to our work are performance studies on incremental view maintenance algorithms. The work in this field is not as extensive as on view maintenance itself, but a number of studies exist. An early paper on measuring the performance of incremental view maintenance strategies is Hanson [Han87]. The ECA paper [ZGMHW95] contains a study on the performance of its algorithm, but only in an analytical manner rather than actual performance studies. [MKK97] also contains a very simple analysis of their algorithm's efficiency and Griffin and Libkin [GL95] give a analytical complexity study of their algorithm but neither paper evaluates system performance.

8 Conclusions

In this paper, we have proposed the first incremental view maintenance algorithm for schema-restructuring views in *SchemaSQL*. We have shown that the traditional approach at incremental view maintenance—rewriting view queries and executing them against the source data—is not feasible for such views. We have defined an update propagation scheme in which updates are propagated from the leaves to the root in the algebra tree corresponding to the query and proved its correctness. Performance experiments on a prototype of a view-maintenance-capable query engine have shown that update propagation has the expected large benefits over recomputation of views. Lastly, we have also proposed a possible performance improvement by the introduction of *batched-update* primitives. In summary, our work reports a significant step towards supporting the integration of large yet schematically heterogeneous data sources into in integrated environment such as a data warehouse, while allowing for incremental propagation of updates.

Future work includes an extension of our implementation towards such batched updates, a review of the implementation in general with the goal of removing some of the performance obstacles (such as the use of local relations to store intermediate data) and an extension of the implementation towards multi-databases. Another direction of research could be the analysis of the concept presented in this paper with regard to its usability for other data integration languages, such as XML query languages or object-relational languages.

References

[AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[BLT86] J. A. Blakeley, P.-E. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *Proceedings of SIGMOD*, pages 61–71, 1986.

[CGL+96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

[CKW89] W. Chen, M. Kifer, and D. Warren. HiLog as a Platform for Database Languages. *IEEE Data Eng. Bull.*, 12(3):37, September 1989.

[FRV95] D. Florescu, L. Raschid, and P. Valduriez. Using Heterogenous Equivalence for Query Rewriting in Multidatabase Systems. In *3rd Int. Conf. on Cooperative Information Systems*, pages 158–169, Vienna, Austria, 1995.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[GLS96] M. Gyssens, L. V. S. Lakshmanan, and I. N. Subramanian. Tables as a Paradigm for Querying and Restructuring (extended abstract). In ACM, editor, *Proceedings of ACM Symposium on Principles of Database Systems*, volume 15, pages 93–103, New York, NY 10036, USA, 1996. ACM Press.

[GLS+97] F. Gingras, L. Lakshmanan, I. N. Subramanian, D. Papoulis, and N. Shiri. Languages for Multi-Database Interoperability. In J. M. Peckman, editor, *Proceedings of SIGMOD*, volume 26(2) of *SIGMOD Record*, pages 536–538, 1997.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[GRVB98] J.-R. Gruser, L. Raschid, M. E. Vidal, and L. Bright. Wrapper Generation for Web-Accessible Data Sources. In *6th Int. Conf. on Cooperative Information Systems*, pages 14–23, New York, 1998.

[Han87] E. N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of SIGMOD*, pages 440–453, 1987.

[HGMI+95] J. Hammer, H. García-Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Information Translation, Mediation, and Mosaic-based Browsing in the TSIMMIS System. In *Proceedings of SIGMOD*, pages 483–483, 1995.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):40–49, June 1991.

[KR01] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views in SchemaSQL. Technical Report WPI-CS-TR-00-25, Worcester Polytechnic Institute, Dept. of Computer Science, January 2001. `http://www.cs.wpi.edu/Resources/techreports`.

[LAZ+89] W. Litwin, A. Abdellatif, A. Zeroual, B. Nicolas, and P. Vigier. MSQL: A Multidatabase Language. *Information Sciences*, 49(1), 1989.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL — A Language for Interoperability in Relational Multi-database Systems. In T. M. Vijayaraman et al., editors, *International Conference on Very Large Data Bases*, pages 239–250, Mumbai, India, Sept. 1996.

[LSS99] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On Efficiently Implementing SchemaSQL on an SQL Database System. In *International Conference on Very Large Data Bases*, pages 471–482, 1999.

[LSS01] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. SchemaSQL–An Extension to SQL for Multidatabase Interoperability. *ACM Transactions on Database Systems*, 26(4):476–519, December 2001.

[Mil98] R. J. Miller. Using Schematically Heterogeneous Structures. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):189–200, 1998.

[MIR93] R. J. Miller, Y. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases (VLDB)*, pages 120–133, Dublin, Ireland, August 1993.

[MKK97] M. K. Mohania, S. Konomi, and Y. Kambayashi. Incremental Maintenance of Materialized Views. In *Database and Expert Systems Applications (DEXA)*, pages 551–560, 1997.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 3(3):337–341, September 1991.

[TAH+96] M. Tork Roth, M. Arya, L. M. Haas, M. J. Carey, W. Cody, R. Fagin, P. M. Schwarz, J. Thomas, and E. L. Wimmers. The Garlic Project. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):557 ff., 1996.

[TRV95] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. Technical report, INRIA, 1995.

[Ull89] J. Ullman. *Principle of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[ZGMHW95] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.